

21 Ant erweitern durch Java-Programmierung

Falls die im Standard vorhandene Funktionalität von Ant für Ihre Bedürfnisse nicht ausreicht, können Sie den Leistungsumfang durch das Programmieren eigener Tasks und Datentypen erweitern. Als Programmiersprache dient natürlich Java. Gegenüber dem Scripting weist diese Variante einige Vorteile auf. Durch Java-Programmierung erreichen Sie eine höhere Performance. Die Weitergabe der selbst programmierten Erweiterungen sowie die Installation auf einem anderen Rechner sind etwas einfacher als beim Scripting, da weniger Dateien beteiligt sind. Außerdem können Sie einige Erweiterungen (z.B. Logger) nur durch Java-Programmierung erstellen, nicht aber durch Scripting.

Die einfache Erweiterbarkeit ermöglicht unterschiedliche Anwendungsszenarien. So existieren schon seit geraumer Zeit Zusatzpakete (z.B. Ant-Contrib, siehe Kapitel 19), die den Funktionsumfang von Ant erweitern. Derartige Pakete müssen explizit installiert werden, damit sie für Ant zur Verfügung stehen. Es gibt aber auch Anwendungen, die ausgewählte Funktionen unter anderem auch als Ant-Task bereitstellen. So existiert innerhalb von Tomcat eine Implementierung des JSP-Compilers als Ant-Task. Auch andere Werkzeuge, z.B. AspectJ, unterstützen inzwischen diese Form des Zugriffs.

Dieses Kapitel beschreibt, wie Sie eigene Ant-Tasks und Ant-Datentypen programmieren und in Ihre Build-Datei einbinden können. Einige der vorgestellten Tasks können Sie während der Einarbeitung in Ant und später während der Entwicklung von Build-Dateien für Debug-Zwecke nutzen.

21.1 Externe Tasks

Externe Tasks müssen in Form einer Class-Datei oder eines Archivs vorliegen. Um einen externen Task aufrufen zu können, muss dieser zunächst deklariert werden. Sie benutzen dazu das `<taskdef>`-Kommando. Dieses Kommando wird üblicherweise mit den Attributen `name` und `classname` sowie ggf. `classpath` verwendet.

Im Attribut `name` legen Sie den Namen fest, unter dem der Task innerhalb der Build-Datei angesprochen werden soll. Der Name ist wahlfrei, wobei Namen existierender Tasks natürlich nicht benutzt werden können. Außerdem müssen Sie die von XML vorgegebenen Einschränkungen berücksichtigen.

Das Attribut `classname` verweist auf die konkrete Klasse, wobei der vollständige Name anzugeben ist, also inklusive einer Package-Bezeichnung. Eventuell müssen Sie zusätzlich auch noch mit dem Attribut `classpath` einen Classpath setzen. Dies ist nicht notwendig, wenn sich die Klasse mit dem Task bereits im Classpath befindet. Das ist z.B. dann der Fall, wenn die zusätzlichen Tasks in einem Jar-Archiv liegen, das sich im `lib`-Verzeichnis von Ant befindet. Wie bei vielen anderen Kommandos auch besteht die Möglichkeit, den Classpath nicht direkt zu notieren, sondern über das Attribut `classpathref` zu referenzieren.

Falls Sie mehrere externe Tasks einbinden möchten, können Sie die Paare aus Taskname und Klassenname auch in eine Property- oder Ressourcendatei auslagern. Außerdem steht ein Attribut zur Verfügung, mit dem Sie einen Classloader angeben können, der das Laden der externen Klassen übernimmt. In Tabelle 21-1 finden Sie die Liste aller möglichen Attribute.

<taskdef>		
Attribut	Beschreibung	Erforderlich
<code>name</code>	Name des neuen Tasks	Ja
<code>classname</code>	Vollständiger Name der Klasse	Ja, falls <code>file</code> oder <code>resource</code> nicht benutzt werden
<code>file</code>	Name einer Property-Datei mit Paaren von <code>name</code> und <code>classname</code>	Nein
<code>resource</code>	Name einer Ressourcendatei mit Paaren von <code>name</code> und <code>classname</code>	Nein
<code>classpath</code>	Classpath zum Zugriff auf die einzubindende Klasse(n)	Nein
<code>classpathref</code>	Referenz auf einen Classpath	
<code>loaderref</code>	Name eines Loaders zum Einlesen des Tasks	Nein

Tab. 21-1 Attribute des `<taskdef>`-Kommandos

21.2 Programmierung

Wenn Sie einen eigenen Task programmieren, müssen Sie dazu auf eine existierende Ant-Klasse aufbauen und diese erweitern. Standard-Tasks erben von `org.apache.tools.ant.Task`. Es gibt weitere Klassen, die für spezielle Tasks benutzt werden müssen. Wir werden hier aber nur auf den Standard-Task eingehen.

Die Basisklasse `org.apache.tools.ant.Task` besitzt bereits alle Funktionen, die Ant benötigt, um den Task in den Build einzubinden und zu initialisieren. Die

minimale Anforderung an einen neuen Task besteht lediglich darin, dass er über eine `execute()`-Methode verfügen muss, in der die eigentliche Funktionalität des Tasks enthalten ist. Bereits mit einem solch einfachen Task sind durchaus nützliche Anwendungen möglich, wie das folgende Beispiel zeigt. Es stellt einen Task vor, der alle Propertyts in sortierter Reihenfolge auflistet.

```
package extensions;
import java.util.*;
import org.apache.tools.ant.*;

public class Echopropsort extends Task {

    public void execute() throws BuildException {
        TreeSet sortedKeys = new TreeSet();
        String key = null;

        Hashtable props = getProject().getProperties();
        Enumeration keys = props.keys();
        while (keys.hasMoreElements())
            sortedKeys.add(keys.nextElement());

        Iterator sortedKeysIterator = sortedKeys.iterator();
        while (sortedKeysIterator.hasNext()) {
            key = (String)sortedKeysIterator.next();
            log(key + " = " + props.get(key));
        }
    }
}
```

Die Methode `execute()` liefert keinen Rückgabewert, kann aber eine Ausnahme `BuildException` auslösen. Mit dieser Ausnahme signalisiert ein Task seiner Umgebung, dass ein Fehler aufgetreten ist. In diesem einfachen Beispiel sind keine Probleme zu erwarten, deshalb wird diese Ausnahme nicht erzeugt.

Alle Tasks werden von Ant innerhalb einer Umgebung, dem Projekt, abgearbeitet. Das Projekt verwaltet eine Vielzahl von Elementen, die von Tasks auch gelesen oder modifiziert werden können. Dazu ist mittels der Methode `getProject()` zunächst eine Referenz auf das aktuelle Projekt zu beschaffen, zu dem der Task gehört. Über diese Referenz ist dann der Zugriff auf die vom Projekt verwalteten Elemente möglich.

Im aktuellen Beispiel liefert die Methode `getProject().getProperties()` eine Liste mit allen Propertyts, die momentan im Projekt existieren. Das Sortieren dieser Liste erfolgt mit Java-Standardtechniken. Lediglich zur Ausgabe auf der Konsole wird wieder eine Ant-Funktion benutzt. Mit der Methode `log(String message)` erzeugen Sie eine Ausgabe auf der Konsole. Ant sorgt dafür, dass der Name des auslösenden Tasks in die Ausgabe aufgenommen wird. Die `log()`-Methode kann optional auch noch einen Log-Level entgegennehmen:

```
log(String message, int logLevel)
```

Sofern beim Aufruf der `log()`-Methode darauf verzichtet wird, benutzt Ant den Log-Level `INFO` als Standardeinstellung.

Das Beispiel wird durch folgende Build-Datei kompiliert und ausgeführt. Für andere Tasks stehen im Download-Archiv ebenfalls Build-Dateien zur Verfügung, die hier aus Platzgründen nicht abgedruckt werden.

```
<project name="bsp2101" default="main" basedir=".">
  <target name="main" depends="compile">
    <taskdef
      name="propsort"
      classname="extensions.Echopropsort"
      classpath="./java/build" />
    <propsort/>
  </target>
  <target name="compile">
    <javac srcdir="./java/src."
      destdir="./java/build"
      includes="extensions/Echopropsort.java" />
  </target>
</project>
```

21.2.1 Attribute

Es gibt nur wenige Fälle, bei denen Tasks ohne zusätzliche Attribute oder eingebettete Tags Sinn machen. Die Übergabe von Attributen an einen Task wird ebenfalls von der Ant-Umgebung übernommen. Der Task muss dazu für jedes Attribut über eine Setter-Methode verfügen, die exakt einen Parameter entgegennimmt. Der Name dieser Methode ergibt sich aus der Zeichenkette `set` und dem Namen des Attributs, wobei das erste Zeichen des Attributnamens großgeschrieben wird. Der Parameter muss nicht vom Typ `String` sein, Ant führt automatisch eine Reihe von Konvertierungen in allgemeine Java-Datentypen oder spezielle Ant-Objekte durch. Die Details entnehmen Sie bitte Unterabschnitt »Typkonvertierungen für Attribute«.

Einfache Attribute

Das nächste Beispiel demonstriert die Übergabe von Parametern an einen eigenen Task. Dabei werden neben der einfachen Übergabe von Zeichenketten auch einige der von Ant durchgeführten Konvertierungen vorgestellt.

Das Beispiel baut auf dem vorangegangenen Task `Echopropsort` auf. Dieser Task wird so erweitert, dass er einen regulären Ausdruck entgegennimmt. Mit diesem Ausdruck können Sie bestimmen, welche Property's der Task auflisten soll. Zur Übergabe des Ausdrucks dient der Parameter `regex`, der unbedingt gesetzt werden muss.

Außerdem soll es möglich sein, die Ausgabe in eine Datei umzuleiten. Der Name der Zieldatei wird in einem zweiten Parameter `file` übergeben. Der Wert eines dritten Parameters `append` legt dabei fest, ob die Datei überschrieben werden soll oder ob die Ausgabe am Ende dieser Datei angefügt wird. Die letzten beiden Attribute sind optional.

Gemäß der Aufgabenstellung müssen drei Setter-Methoden programmiert werden. Diese Methoden setzen jeweils ein privates Attribut innerhalb der neuen Klasse. Das Listing zeigt, dass nicht alle Attribute vom Typ `String` sind. Lediglich der reguläre Ausdruck wird als `String` abgelegt. Für den `append`-Parameter wird ein `boolean`-Feld benutzt, für die Ausgabedatei verwendet der Task ein `File`-Objekt.

Interessant ist in diesem Zusammenhang der Quelltext der drei Setter-Methoden. Ant sorgt dafür, dass den Setter-Methoden bereits ein Objekt übergeben wird, das den von Ihnen gewünschten Typ besitzt. Der reguläre Ausdruck wird als `String` übergeben, für die Datei erzeugt Ant das notwendige `File`-Objekt und im Falle des booleschen Wertes wandelt Ant die in der Build-Datei zu benutzenden Werte (`true`, `yes`, `ok` bzw. `false`, `no`, `off`) in den korrekten logischen Wert um.

Ein weiterer Unterschied zum ersten Beispiel besteht in der Prüfung der Attributwerte. Nicht in jedem Fall benötigt ein Task alle Attribute. Mitunter gibt es Abhängigkeiten der Attribute untereinander oder die Verwendung bestimmter Attribute schließt sich gegenseitig aus. In vielen Fällen ist auch die Überprüfung des konkreten Wertes eines Parameters erforderlich. Derartige Prüfungen müssen Sie komplett selbst programmieren. Üblicherweise erfolgt dies durch eine eigene Methode, die unmittelbar am Anfang der `execute()`-Methode aufgerufen wird. In diesem Beispiel ist das die Methode `checkParams()`. Sie führt hier zwei Überprüfungen aus. Zunächst wird geprüft, ob der Parameter `regex` überhaupt gesetzt wurde. Wenn ja, erfolgt durch eine Standard-Java-Klasse ein Syntaxcheck des regulären Ausdrucks. Falls die Überprüfungen nicht zufrieden stellend verlaufen, wird eine `BuildException` ausgelöst. Dem Konstruktor dieser Ausnahmeklasse können bzw. sollten Sie einen erläuternden Text übergeben.

```
package extensions;
import java.io.*;
import java.util.*;
import java.util.regex.*;
import org.apache.tools.ant.*;

public class Echoregexp extends Task {
    private String regex = null;
    private File outf = null;
    private boolean append = false;

    public void setRegexp(String regex) {
        this.regex = regex;
    }
}
```

```
public void setAppend(boolean app){
    this.append = app;
}

public void setFile(File outf) {
    this.outf = outf;
}

public void execute() throws BuildException {
    FileWriter out = null;
    checkParams();

    Hashtable props = project.getProperties();
    Enumeration keys = props.keys();

    TreeSet sortedKeys = new TreeSet();

    String key = null;

    while (keys.hasMoreElements()) {
        key = (String)keys.nextElement();
        if (Pattern.matches(regex, key)) {
            sortedKeys.add(key);
        }
    }

    Iterator filteredKeys = sortedKeys.iterator();

    if (outf != null){
        try {
            out = new FileWriter(outf, append);
        } catch (IOException ioe) {
            throw new BuildException(ioe);
        }
    }

    String msg = null;
    while (filteredKeys.hasNext()) {
        key = (String)filteredKeys.next();
        msg = key + " = " + props.get(key);
        if (out == null) {
            log(msg);
        }
        else {
            try {
                out.write(msg + System.getProperty("line.separator"));
            }
            catch (IOException ioe) {
                throw new BuildException(ioe);
            }
        }
    }
}
```

```
        if (out != null) {
            try {
                out.close();
            }
            catch (IOException ioe) {
                throw new BuildException(ioe);
            }
        }
    }

    private void checkParams() {
        if (regex == null) {
            throw new BuildException("Attribute regexp must be set");
        }
        try {
            Pattern.compile(regex);
        }
        catch (PatternSyntaxException pse) {
            throw new BuildException(pse.getMessage());
        }
    }
}
```

Aufzählungsattribute

In Ant gibt es viele Beispiele dafür, dass ein Attribut nur einen Wert aus einer vorgegebenen Menge annehmen kann. Beispiele für solche Attribute sind der Log-Level im `<echo>`-Kommando, die Zeiteinheit im `<waitfor>`-Kommando oder die Auswahlmöglichkeiten für das Zeilenvorschubzeichen im `<fixCrLf>`-Kommando.

Ant bietet eine Unterstützung für derartige Aufzählungsattribute in Form einer abstrakten Klasse `EnumeratedAttribute` aus dem Package `org.apache.tools.ant.types`. Sie müssen diese Klasse auf jeden Fall überschreiben, wobei unterschiedlich komplexe Szenarien denkbar sind.

Die Funktionen der Klasse `EnumeratedAttribute` und der von ihr abgeleiteten Klassen sind relativ einfach. Wichtig ist zunächst, dass für das Aufzählungsattribut im Task eine Setter-Methode existiert, die ein Objekt einer von `EnumeratedAttribute` abgeleiteten Klasse entgegennimmt. Wenn Ant erkennt, dass einer Setter-Methode ein solches Objekt übergeben werden soll, erzeugt Ant zunächst ein solches Aufzählungsobjekt. Das ist sehr einfach, da der Konstruktor von `EnumeratedAttribute` keinerlei Parameter erwartet. Anschließend ruft Ant die `setValue()`-Methode des Aufzählungsobjektes auf. Diese Methode ist in `EnumeratedAttribute` vollständig implementiert. Sie erwartet einen String als Parameter. Ant übergibt an dieser Stelle den Wert des XML-Attributs aus der Build-Datei. Die `setValue()`-Methode prüft, ob der konkrete Wert in der Menge der zulässigen Werte enthalten ist. Dazu ruft sie die Methode `getValues()` der neu programmierten Aufzählungsklasse auf. Diese Methode liefert alle zulässigen Werte als ein Array von

Strings zurück. Sie müssen diese Methode auf jeden Fall überschreiben, um die Wertemenge des Attributs festzulegen. Verläuft die Prüfung korrekt, wird der Attributwert von `setValue()` in einer privaten Variablen des Aufzählungsobjektes abgelegt. Er kann jederzeit mittels der `getValue()`-Methode gelesen werden.

Für den einfachsten Einsatzfall reicht es aus, die zulässige Wertemenge als String-Array bereitzustellen. Ant kann dann zur Laufzeit prüfen, ob der aktuelle Wert des Attributs in dieser Wertemenge enthalten ist. Diese einfache Variante lohnt sich aber meist nur dann, wenn der Attributwert ein String ist, der im Task unverändert weiter verwendet werden soll. Andernfalls müsste innerhalb des Tasks eine Fallunterscheidung durchgeführt werden, um den Programmablauf vom Attributwert abhängig zu machen.

Wesentlich häufiger muss der Attributwert in irgendeiner Form ausgewertet oder »übersetzt« werden. Die eingangs genannten Beispiele deuten dies bereits an. Die innerhalb von Ant zur Konsolenausgabe verwendete `log()`-Methode erwartet den Log-Level als Integerwert und nicht als String. Im `<echo>`-Kommando muss der Wert des `level`-Attributs daher in einen Integerwert umgesetzt werden. Dies kann z.B. durch eine Reihe verschachtelter `if`-Anweisungen innerhalb der `execute()`-Methode geschehen. Etwas eleganter ist es, die Konvertierung in die Aufzählungsklasse zu verlagern.

Zur Demonstration sollen anschließend beide Varianten anhand der einfachsten Version der Echopropsort-Klasse vorgestellt werden. Ziel ist es, die von diesem Task erzeugten Ausgaben mit einem Log-Level zu versehen.

Direkte Verwendung des Attributwertes

Im folgenden Listing finden Sie die Aufzählungsklasse `LogLevel` als Inner Class direkt am Beginn der neuen Klasse.

Einzigster Bestandteil der neuen Aufzählungsklasse ist die Methode `getValues()`. Diese Methode liefert die zulässigen Werte zurück. Sie werden hart im Quelltext kodiert.

Die Setter-Methode für das `LogLevel`-Attribut nimmt ein Objekt vom Typ `LogLevel` entgegen. Ant erkennt zur Laufzeit, dass ein Objekt vom Typ `LogLevel` erforderlich ist. Es erstellt das Objekt und weist den Attributwert zu. Die Aufzählungsklasse selbst prüft dessen Gültigkeit.

Mit der Methode `getValue()` können Sie den Wert ermitteln, mit dem ein Aufzählungsattribut initialisiert wurde. In der hier gezeigten einfachsten Variante der Aufzählungsattribute wird dieser Wert in einer Variablen `level` zwischengespeichert. In der `execute()`-Methode erfolgt dann die Umsetzung der Zeichenkette in einen echten Log-Level-Wert.

Die Aufzählungsklasse sorgt somit nur für eine Prüfung des Attributwertes. Der diesbezügliche Nutzen hält sich in Grenzen, da umfangreiche Erweiterungen im eigentlichen Task notwendig sind.

```
package extensions;
import java.util.*;
import org.apache.tools.ant.*;
import org.apache.tools.ant.types.EnumeratedAttribute;

public class EchoproplogSimple extends Task {

    public static class LogLevel extends EnumeratedAttribute {
        public String[] getValues() {
            return new String[] {
                "error",
                "warning",
                "info",
                "verbose",
                "debug"
            };
        }
    }

    private String level = null;

    public void setLogLevel(LogLevel level) {
        this.level = level.getValue();
    }

    public void execute() throws BuildException {
        int logLevel = Project.MSG_INFO;
        if (level.equals("debug"))
            logLevel = Project.MSG_DEBUG;
        else if (level.equals("error"))
            logLevel = Project.MSG_ERR;
        else if (level.equals("info"))
            logLevel = Project.MSG_INFO;
        else if (level.equals("verbose"))
            logLevel = Project.MSG_VERBOSE;
        else if (level.equals("warning"))
            logLevel = Project.MSG_WARN;

        Hashtable props = getProject().getProperties();
        Enumeration keys = props.keys();

        TreeSet sortedKeys = new TreeSet();

        while (keys.hasMoreElements())
            sortedKeys.add(keys.nextElement());

        Iterator filteredKeys = sortedKeys.iterator();

        while (filteredKeys.hasNext()) {
            String key = (String)filteredKeys.next();
            log(key + " = " + props.get(key), logLevel);
        }
    }
}
```

Transformation des Attributwertes

Der Nutzen eines Aufzählungsattributs kann wesentlich erhöht werden, wenn die Umwandlung des Attributwertes in den endgültigen Wert direkt von der Aufzählungsklasse übernommen werden würde. Dies erfordert etwas mehr Programmierarbeit in dieser Klasse, vereinfacht aber den Code des eigentlichen Tasks erheblich.

Da der Code der Aufzählungsklasse etwas komplexer ist, soll diese Klasse zunächst separat beschrieben werden. Diese Klasse wird im weiteren Verlauf des Beispiels als Inner Class in eine andere Klasse eingebunden.

Zunächst erbt die Aufzählungsklasse natürlich wieder von `EnumeratedAttribute`.

```
public static class LogLevel extends EnumeratedAttribute {
```

Es schließen sich einige Konstantendeklarationen für die möglichen Attributwerte an. Diese Deklarationen vereinfachen den nachfolgenden Code, da die Attributwerte an mehreren Stellen benutzt werden.

```
    private static final String DEBUG = "debug";
    private static final String ERR = "error";
    private static final String INFO = "info";
    private static final String VERBOSE = "verbose";
    private static final String WARN = "warning";
```

Der Einfachheit halber kann auch der Rückgabewert der `getValues()`-Methode als Konstante deklariert werden. Die Nähe zu den anderen Konstanten erleichtert spätere Ergänzungen.

```
    private static final String[] levels = {
        DEBUG, ERR, INFO, VERBOSE, WARN
    };
```

Natürlich muss die `getValues()`-Methode überschrieben werden. Da der Rückgabewert bereits als Konstante vorgefertigt wurde, ist der Code sehr einfach.

```
    public String[] getValues() {
        return levels;
    }
```

Die Umsetzung der Attributwerte in Log-Level-Werte kann in diesem Fall am einfachsten durch eine »Übersetzungstabelle« erfolgen. Eine Hashtabelle nimmt den Attributwert als Schlüssel und den LogLevel als dazugehörigen Wert auf. Diese Tabelle ist zunächst zu deklarieren. Gefüllt wird sie im Konstruktor der Klasse.

```

private Hashtable logLevels = new Hashtable();
public Loglevel() {
    logLevels.put(DEBUG, new Integer( Project.MSG_DEBUG));
    logLevels.put(ERR, new Integer( Project.MSG_ERR));
    logLevels.put(INFO, new Integer( Project.MSG_INFO));
    logLevels.put(VERBOSE, new Integer( Project.MSG_VERBOSE));
    logLevels.put(WARN, new Integer( Project.MSG_WARN));
}

```

Zu guter Letzt die wichtigste Methode der neuen Klasse: `getLoglevel()`. Sie liefert den zu einem Attributwert gehörenden Log-Level als Integerwert zurück. Im Task rufen Sie später diese Methode auf. Da die Attributklasse mit einem der möglichen Attributwerte initialisiert wird, erwartet die `getLoglevel()`-Methode keinen Parameter. Sie greift vielmehr mittels der bereits erwähnten Methode `getValue()` auf den bereits gespeicherten Attributwert zu und liest das dazu passende Objekt aus der Hashtabelle.

```

public int getLoglevel() {
    String key = getValue().toLowerCase();
    Integer i = (Integer) logLevels.get(key);
    return i.intValue();
}
}

```

Die Task-Klasse, genauer gesagt deren `execute()`-Methode, wird nun wieder etwas einfacher. In der Setter-Methode für das `loglevel`-Attribut können Sie nun anstelle der `getValue()`-Methode die neu geschaffene Methode `getLoglevel()` verwenden. Der Log-Level wird in einer privaten Variablen aufbewahrt. Die Initialisierung mit einem Defaultwert stellt sicher, dass der Task auch ohne das Attribut `loglevel` aufgerufen werden kann.

```

package extensions;
import java.util.*;
import org.apache.tools.ant.*;
import org.apache.tools.ant.types.EnumeratedAttribute;

public class EchopropsortAdvanced extends Task {
    private int loglevel = Project.MSG_INFO;

    public void setLoglevel(Loglevel level) {
        loglevel = level.getLoglevel();
    }

    public void execute() throws BuildException {
        Hashtable props = getProject().getProperties();
        Enumeration keys = props.keys();

        TreeSet sortedKeys = new TreeSet();

        while (keys.hasMoreElements())
            sortedKeys.add(keys.nextElement());
    }
}

```

```

Iterator filteredKeys = sortedKeys.iterator();
while (filteredKeys.hasNext()) {
    String key =(String)filteredKeys.next();
    log(key + " = " + props.get(key), loglevel);
}
}
}

public static class Loglevel extends EnumeratedAttribute {
    ...
}
}

```

Typkonvertierungen für Attribute

Während der Ausführung eines Projektes untersucht Ant die Setter-Schnittstellen jedes Tasks. Der Sinn liegt darin, den Attributwert in den vom Task gewünschten Datentyp zu wandeln und dem Task diese Arbeit zu ersparen. Dabei führt Ant einige sinnvolle Konvertierungen durch.

Erwarteter Datentyp	Ant-Aktion
boolean	Wenn der Attributwert true, yes oder on ist, wird der boolesche Wert true übergeben, sonst false.
char oder java.lang.Character	Das erste Zeichen des Attributwerts wird übergeben.
Andere einfache Datentypen (z.B. int)	Der Attributwert wird, wenn möglich, in den Zieltyp umgewandelt. Bei Konvertierungsproblemen wird BuildException ausgelöst.
java.io.File	Ein File-Objekt wird erzeugt. Der Attributwert wird als Dateiname (inklusive Pfad) benutzt. Relative Pfade beziehen sich auf das aktuelle Arbeitsverzeichnis
org.apache.tools.ant.types.Path	Der Attributwert wird gemäß der Trennzeichen »:« und »;« aufgespalten. Jeder Bestandteil wird ein Element des Path-Objektes.
java.lang.Class	Der Attributwert wird als Klassenname benutzt und diese Klasse mit dem System Class Loader geladen.
Andere Klassen, die einen Konstruktor mit einem einzelnen String als Parameter besitzen	Eine Instanz der Klasse wird erzeugt. Der Attributwert wird dem Konstruktor übergeben.
Eine Sub-Klasse von org.apache.tools.ant.types.EnumeratedAttribute	Eine Instanz der Klasse wird erzeugt. Anschließend wird die Methode setValue(String s) dieser Instanz aufgerufen, wobei der Attributwert als Parameter übergeben wird.

Tab. 21-2 Implizite Konvertierungen für Setter-Methoden

21.2.2 Eingebettete Tags

In einen selbst geschriebenen Task können Sie natürlich auch Sub-Tags einbetten. Dabei kann es sich sowohl um Ant-Datentypen (z.B. Filesets) als auch Ant-Tasks handeln. Dazu sind unterschiedliche Mechanismen notwendig. Außerdem muss unterschieden werden zwischen der Verwendung vorhandener Elemente und selbst geschriebenen Datentypen. Dieser Abschnitt beschreibt alle Aspekte.

Ant-Datentypen in eigene Tasks aufnehmen

Jedes Sub-Tag der Build-Datei wird innerhalb Ihres Tasks durch ein eigenes Objekt abgebildet. Filesets und Mapper dürften relativ häufig notwendig sein, um Ihrem Task die einfache Auswahl von Dateien zu ermöglichen. Eingebunden werden Sub-Tags ähnlich wie Attribute. Sie schreiben eine Methode, deren Namen sich aus einem vorgegebenen Präfix und dem Namen des Tags zusammensetzt. Dabei wird der erste Buchstabe des Tag-Namens großgeschrieben.

Im Gegensatz zur Behandlung der Attribute gibt es für das Einfügen von Sub-Tags drei verschiedene Möglichkeiten und damit drei unterschiedliche Java-Methoden. Sie sollten innerhalb eines Tasks für jede Art von Sub-Tag natürlich nur eine Methode programmieren. Andernfalls ist nicht definiert, welche der Methoden zur Laufzeit wirklich ausgeführt wird.

Die drei Varianten unterscheiden sich vor allem dadurch, ob das mit dem Sub-Tag korrespondierende Objekt außerhalb Ihres Tasks von Ant oder innerhalb des Tasks von Ihnen selbst instanziiert wird. Für den erstgenannten Fall existiert dann noch die Unterscheidung, ob dem Tasks das instanziierte, aber noch nicht initialisierte Objekt übergeben wird oder ob Ant das Objekt komplett initialisiert, also alle Parameter übergibt und eventuelle Sub-Tags erzeugt usw. Auch wenn Sie das Sub-Objekt selbst erzeugen, wird es später von Ant initialisiert. Um das Setzen von Parametern oder die Behandlung von Sub-Tags im Sub-Tag müssen Sie sich also nicht kümmern.

Die beiden letzten Varianten können Sie nur für Klassen mit parameterlosem Konstruktor verwenden. Tabelle 21–3 erläutert den Aufbau der Methodennamen und deren Parameter.

Methode	Beschreibung
<code>public NestedElement createTagName()</code>	Das zum Sub-Tag passende Objekt vom Typ <code>NestedElement</code> wird innerhalb der Methode erzeugt und als Rückgabewert übergeben.
<code>public void addTagName(NestedElement element)</code>	Ein instanziiertes, aber noch nicht initialisiertes <code>Element</code> wird von Ant an die Methode übergeben.
<code>public void addConfiguredTagName(NestedElement element)</code>	Ant übergibt ein vollständig initialisiertes <code>Element</code> .

Tab. 21–3 Varianten zum Einfügen von Sub-Tags

Das folgende Beispiel demonstriert zunächst auf einfachste Weise das Prinzip. Der neue Task akzeptiert ein oder mehrere Filesets als eingebettete Objekte. Er listet die Namen aller selektierten Dateien auf der Konsole auf.

```
package extensions;
import org.apache.tools.ant.Task;
import java.util.*;
import org.apache.tools.ant.types.*;
import org.apache.tools.ant.*;

public class FilesetList extends Task {
    private Vector filesets = new Vector();

    public void addConfiguredFileSet(FileSet fs) {
        filesets.add(fs);
    }

    public void execute()throws BuildException {
        for(int i=0; i < filesets.size(); i++) {
            DirectoryScanner ds =
                ((FileSet)filesets.get(i)).getDirectoryScanner(project);
            String files[] = ds.getIncludedFiles();
            for (int k=0; k < files.length; k++)
                log((String)files[k]);
        }
    }
}
```

Im Task wird zunächst ein leerer Vector `filesets` erzeugt, der die Referenzen auf alle Filesets aufnimmt. Bezüglich der Einbindung der Sub-Tags wurde die Variante gewählt, bei der die komplett konfigurierte Klasse an den Tasks übergeben wird. Die dafür zuständige Methode erhält daher den durch das Programmiermodell vorgegebenen Namen `addConfiguredFileset()`. Sie muss lediglich das Fileset-Objekt in den erwähnten Vector einfügen.

Das Fileset-Objekt wird von Ant initialisiert. Das bedeutet, dass Ant die Build-Datei analysiert und alle Attribute sowie weitere Sub-Tags unterhalb des Fileset-Tags behandelt.

Innerhalb der `execute()`-Methode verfügen Sie daher über korrekt parametrisierte Filesets, die Sie nur noch Auslesen müssen. Dies erfolgt mit Hilfe eines so genannten `DirectoryScanners`, der ein Bestandteil eines Filesets ist. Details zu den Klassen `FileSet` und `DirectoryScanner` finden Sie im Abschnitt 20.3.2.

21.2.3 Sub-Tasks

Normalerweise sind Ant-Tasks eigenständige Elemente, die auf der ersten Ebene unterhalb eines Targets notiert werden. Allerdings gibt es einige Tasks zur Ablaufsteuerung, die selbst wieder mehrere Tasks zusammenfassen. Genannt sei

hier nur das `<waitfor>`-Kommando oder das `<parallel>`-Kommando. Auch solche Tasks können Sie programmieren.

Das Prinzip ist relativ einfach. Ebenso wie bei den Attributen oder den Ant-Datentypen übernimmt die Laufzeitumgebung von Ant den größten Teil der Arbeit. Ein Task, der andere Tasks enthalten kann, muss das Interface `TaskContainer` implementieren. Dieses Interface definiert die Methode `addTask()`, die Sie in Ihrer Klasse implementieren müssen. Ebenso wie bei den anderen Elementen eines Tasks müssen Sie alle Sub-Tasks in einer Liste speichern und in der `execute()`-Methode für deren Abarbeitung sorgen. Um ein einfaches, aber auch nützliches Anwendungsbeispiel zu geben, soll eine If-Anweisung nachgebildet werden. Ein `<if>`-Task fasst mehrere andere Tasks zusammen und führt sie nur aus, wenn der Wert des Attributs `condition` dem logischen Wert `true` entspricht. Der Quelltext ist so einfach, dass er keiner weiteren Erläuterung bedarf.

```
package extensions;
import org.apache.tools.ant.Task;
import org.apache.tools.ant.TaskContainer;
import java.util.Vector;

public class MyIf extends Task implements TaskContainer{
    private Vector tasks = new Vector();
    private boolean doit = false;

    public void setCondition(boolean doit){
        this.doit = doit;
    }

    public void execute(){
        if (doit){
            for(int i=0; i < tasks.size(); i++){
                Task t = (Task)tasks.get(i);
                t.perform();
            }
        }
    }

    public void addTask(Task task) {
        tasks.add(task);
    }
}
```

21.2.4 Eigene Datentypen

Beim Programmieren eigener Tasks entsteht oft der Bedarf nach individuellen Sub-Tags, die für die Verwendung in Ihrem eigenen Task zugeschnitten sind. Hier stehen Ihnen zwei unterschiedliche Programmiermodelle zur Verfügung.

Zunächst können Sie das Sub-Tag als eingebettete Klasse innerhalb Ihres Tasks programmieren. Dies vereinfacht die spätere Anwendung, da kein zusätz-

liches `<typedef>`-Kommando erforderlich ist, um das Sub-Tag bekannt zu machen. Allerdings können Sie dieses Sub-Tag dann nur im Zusammenhang mit diesem einen Task einsetzen. Außerdem können Sie zum Einfügen des Sub-Tags in Ihren Task nur die `createTagName()`-Methoden benutzen.

All diese Einschränkungen können Sie umgehen, indem Sie für das Sub-Tag eine eigenständige Klasse erzeugen. Dieses Vorgehen erfordert allerdings den Einsatz des `<typedef>`-Kommandos in der Build-Datei. Außerdem muss der Name, unter dem das Sub-Tag in der Build-Datei verfügbar ist, dem Namen entsprechen, der im umgebenden Task erwartet wird. Er kann somit nicht mehr frei gewählt werden.

Für das erstgenannte Programmiermodell soll hier ein Beispiel vorgestellt werden. Es wird ein neuer Task erstellt, der einen Swing-Dialog einblendet. Dieser Dialog ermöglicht per Checkboxes die Auswahl unter einigen Optionen. Die Definition der Checkboxes erfolgt über Sub-Tags.

In einer Build-Datei könnten Sie mit einem solchen Dialog Varianten des Builds wählen oder seltener benötigte zusätzliche Aufgaben aktivieren.

Der Quellcode ist wiederum recht einfach gehalten und ordnet sich der Aufgabe unter, das Prinzip zu demonstrieren. Zur Verdeutlichung der Anwendung dieses Tasks hier zunächst die Build-Datei. Im Dialog können Sie beliebig viele der Checkboxes markieren. Im Anschluss werden die Namen der zugehörigen Propertys aufgelistet.

```
<project name="bsp2107" default="main" basedir=".">
  <target name="main" depends="compile">
    <taskdef
      name="checkdialog"
      classname="dialogs.CheckboxDialog"
      classpath="./java/build" />
    <checkdialog>
      <checkoption
        property="cb.p1"
        label="Label Auswahl 1"
        tooltip="Auswahl 1" />
      <checkoption
        property="cb.p2"
        label="Label Auswahl 2"
        tooltip="Auswahl 2"/>
      <checkoption
        property="cb.p3"
        label="Label Auswahl 3"
        tooltip="Letzte Auswahlmöglichkeit" />
    </checkdialog>
    <echoproperties prefix="cb"/>
  </target>
```

```
<target name="compile">
  <javac
    srcdir="./java/src"
    destdir="./java/build"
    includes="dialogs/CheckboxDialog.java"/>
</target>
</project>
```

Kernstück ist das Target `main`. Dieses Target macht den noch zu erläuternden Dialog-Task unter dem Namen `checkdialog` bekannt. Der Task definiert mittels des `<checkoption>`-Sub-Tags drei Auswahlfelder. Nach Ausführung des Dialogs werden per `<echoproperties>`-Task die ggf. erstellten Property's angezeigt.

Nun zum Java-Quelltext. Für eine übersichtlichere Beschreibung finden Sie den Quelltext wieder auf mehrere Abschnitte verteilt. Schwerpunkt sind die Ant-spezifischen Teile, nicht jedoch die Programmierung eines Swing-Dialogs.

Zu Beginn der Datei werden die notwendigen Java-Packages importiert.

```
package dialogs;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import org.apache.tools.ant.*;
```

Es schließt sich die Deklaration der Klasse für den neuen Task an. Da ein Swing-Dialog erzeugt werden soll, wird die Implementierung des `ActionListener`-Interfaces notwendig.

```
public class CheckboxDialog
  extends Task
  implements ActionListener {
```

Einige Attribute der neuen Klasse dienen zur Aufnahme der diversen Optionen.

```
  private String property = null;
  private JDialog jd = null;
  private JButton bOK = null;
  private JButton bCancel = null;
  private Vector optionlist = new Vector();
  private boolean dialogStatus = false;
```

Die einzelnen Auswahlmöglichkeiten werden über Sub-Tags definiert. Dazu muss die neue Klasse über eine entsprechende `create()`-Methode verfügen. Da die Klasse für das Sub-Tag als Inner Class realisiert wird, muss die Instanziierung innerhalb der `create()`-Methode erfolgen und kann nicht von Ant übernommen werden. Für jede Auswahlmöglichkeit wird eine separate Instanz der `Checkoption`-Klasse in einer Liste aufbewahrt. Die `Checkoption`-Klasse wird weiter unten beschrieben.

```
public Checkoption createCheckoption(){
    Checkoption co = new Checkoption();
    optionlist.add(co);
    return co;
}
```

Der Hauptteil der Arbeit wird in der `execute()`-Methode ausgeführt. In dieser Methode wird der Swing-Dialog erstellt.

```
public void execute() throws BuildException {
    jd = new JDialog();
    jd.getContentPane().setLayout(new BorderLayout(25, 25));
    jd.setLocation(400,400);

    JPanel p1 = new JPanel();
    p1.setLayout(new GridLayout(optionlist.size(), 1));
```

Jede `Checkoption`-Klasse erstellt selbst eine `Checkbox` und fügt diese in den `Swing-Dialog` ein.

```
for(int n=0; n<optionlist.size(); n++){
    Checkoption co = (Checkoption)optionlist.get(n);
    p1.add(co.jcb);
}
```

Es schließen sich weitere Aktivitäten zur Gestaltung des `Swing-Dialogs` an.

```
JPanel p2 = new JPanel();
p2.setLayout(new FlowLayout(FlowLayout.CENTER, 25, 25));

bOK = new JButton("OK");
bOK.addActionListener(this);
p2.add(bOK);

bCancel = new JButton("Cancel");
bCancel.addActionListener(this);
p2.add(bCancel);

jd.getContentPane().setLayout(new GridLayout(2,1));
jd.getContentPane().add(p1);
jd.getContentPane().add(p2);
jd.setSize(300,300);
jd.setModal(true);
jd.pack();
```

Letztendlich wird der Dialog als modaler Dialog ausgeführt. Die `execute()`-Methode wartet, bis die `show()`-Methode zurückkehrt und wertet dann den Rückgabewert des Dialogs aus, der in der `actionPerformed()`-Methode gesetzt wird. Falls der Dialog abgebrochen wurde, wirft der `Task` eine `Build-Exception` und beendet dadurch den `Build`.

```
        jd.show();
        if (dialogStatus == false) {
            throw new BuildException("Canceled by user");
        }
    }
}
```

Da die Behandlung von Swing-Dialogen eine spezielle Programmieretechnik erfordert, muss die Auswertung der Dialoginformationen in einer separaten Methode erfolgen.

```
public void actionPerformed(ActionEvent ae) {
    Object source = ae.getSource();
    if (source == bOK) {
        Component[] cpl = jd.getComponents();
        for(int i=0; i < optionlist.size();i++){
            Checkoption co = (Checkoption)optionlist.get(i);
            if(co.jcb.isSelected())
                this.getProject().setProperty(co.getProperty(), "true");
        }
        dialogStatus = true;
    }
    jd.setVisible(false);
    jd.dispose();
}

public class Checkoption {
    // siehe weiter unten
}
}
```

Die Checkoption-Klasse ist recht einfach. Sie enthält als Member eine Checkbox, die mittels der beiden Attribute `label` und `tooltip` parametrisiert werden kann. Falls die entsprechende Checkbox selektiert wurde, wird ein Property mit dem Wert `true` angelegt. Der Name des anzulegenden Property steht im Attribut `property`.

```
public class Checkoption {
    private JCheckBox jcb = new JCheckBox();

    public void setLabel(String l){
        jcb.setLabel(l);
    }

    public void setTooltip(String t){
        jcb.setToolTipText(t);
    }

    public void setProperty(String p){
        jcb.setActionCommand(p);
    }
}
```

```
public String getProperty(){
    return jcb.getActionCommand();
}
}
```

Eine umfangreichere Implementierung eines Dialogs für Ant finden Sie im Download-Archiv. Dort stehen neben Checkboxes auch Radiobuttons und Textfelder sowie eine zusätzliche Gruppierungsmöglichkeit zur Verfügung. Außerdem können Sie die Eingabelemente auch ineinander verschachteln.